

## AI Assisted Coding

S.AKSHITH REDDY

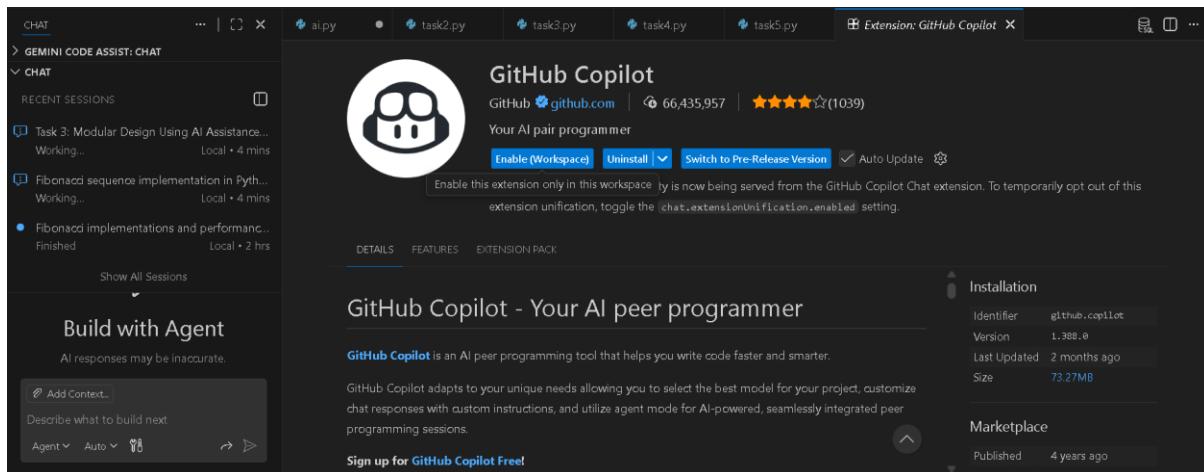
2303A51383

BATCH - 06

## Lab 1: Environment Setup – GitHub Copilot and VS Code Integration + Understanding AI-assisted Coding Workflow

### Lab Objectives:

#### Week1 -



### Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence

#### Without Functions)

##### PROMPT:

Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence

Without Functions)

##### ❖ Scenario

You are asked to write a quick numerical sequence generator for a learning platform prototype.

##### ❖ Task Description

Use GitHub Copilot to generate a Python program that:

- Prints the Fibonacci sequence up to n terms
  - Accepts user input for n
  - Implements the logic directly in the main code
  - Does not use any user-defined functions

❖ Expected Output

  - Correct Fibonacci sequence for given n
  - Screenshot(s) showing Copilot-generated suggestions
  - Sample inputs and outputs

The screenshot shows a code editor with a Python file named `ai.py` open. The code is a simple script to generate a Fibonacci sequence up to `n` terms. It includes comments explaining the task, requirements, and logic. The code uses `input()` to get user input for the number of terms. The terminal below shows the script running and outputting the sequence for 5 terms.

```
ai.py x task2.py task3.py task4.py task5.py cp.py
ai.py > ...
1 '''Task 1: AI-Generated Logic Without Modularization (Fibonacci Sequence
2 Without Functions)
3 ♦ Scenario
4 You are asked to write a quick numerical sequence generator for a learning
5 platform prototype.
6 '''
7
8 # Fibonacci sequence up to n terms
9 # Requirements:
10 # - Accepts user input for n
11 # - Implements logic directly in the main code (no user-defined functions)
12 # - Prints the Fibonacci sequence up to n terms
13
14 if __name__ == "__main__":
15     try:
16         n_input = input("Enter number of terms: ").strip()
17         n = int(n_input)
18         if n <= 0:
19             print("Please enter a positive integer.")
20         else:
21             print(f"Fibonacci sequence: {fibonacci(n)}")
22
PROBLEMS DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER
PS C:\Users\preet\Downloads\pythonbasics> & C:/Users/preet/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/preet/Downloads/pythonbasics/
cp.py
PS C:\Users\preet\Downloads\pythonbasics> & C:/Users/preet/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/preet/Downloads/pythonbasics/
ai.py
Enter number of terms: 5
Fibonacci sequence: 0 1 1 2 3
PS C:\Users\preet\Downloads\pythonbasics>
```

**OUTPUT:** Enter number of terms: 5

Fibonacci sequence: 0 1 1 2 3

## Code Explanation (Fibonacci without functions)

- `input()` → takes number of terms from user
  - `int()` → converts input to integer
  - if  $n \leq 0$  → checks for invalid (non-positive) input

- $a, b = 0, 1 \rightarrow$  starting Fibonacci numbers
- `for _ in range(n)`  $\rightarrow$  runs loop n times
- `seq.append(str(a))`  $\rightarrow$  stores current Fibonacci number
- $a, b = b, a + b \rightarrow$  updates values for next term
- `" ".join(seq)`  $\rightarrow$  prints numbers in one line
- `try / except`  $\rightarrow$  handles invalid (non-integer) input safely

**Output:** Prints Fibonacci sequence up to n terms.

## Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

### PROMPT: Task 2: AI Code Optimization & Cleanup (Improving Efficiency)

#### ❖ Scenario

The prototype will be shared with other developers and needs optimization.

#### ❖ Task Description

- Examine the Copilot-generated code from Task 1 and improve it by:
  - Removing redundant variables
  - Simplifying loop logic
  - Avoiding unnecessary computations
  - Use Copilot prompts such as:
    - “Optimize this Fibonacci code”
    - “Simplify variable usage”

Hint:

Prompt Copilot with phrases like

“optimize this code”, “simplify logic”, or “make it more readable”

#### ❖ Expected Output

- Original vs improved code
- Written explanation of:
  - What was inefficient

- How the optimized version improves performance and readability

```

 1  """Task 2 - Optimization & Cleanup: Fibonacci
 2
 3 This file contains:
 4 - The original Copilot-style implementation (as `original_fib_list`) - builds string list inside loop.
 5 - The improved implementation (as `optimized_fib_generator`) - yields integers and separates generation from presentation.
 6 - A small benchmark that compares time and peak memory for the two approaches.
 7
 8 Use: python task2.py          # runs a small demo
 9      python task2.py --bench   # runs a simple benchmark (default n=10000)
10 """
11 from __future__ import annotations
12
13 import time
14 import tracemalloc
15 import sys
16 from typing import Generator, Iterable, List
17
18
19 # ----- Original (Copilot-generated) -----
20 # Inefficient scenario:
PROBLEMS DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER
PS C:\Users\preet\Downloads\pythonbasics> & C:/Users/preet/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/preet/Downloads/pythonbasics/
task2.py
Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34

Optimized output:
Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34
ps C:\Users\preet\Downloads\pythonbasics>

```

## OUTPUT:

Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34

Optimized output:

Fibonacci sequence: 0 1 1 2 3 5 8 13 21 34

## CODE EXPLANATION

### Optimization & Cleanup (Fibonacci)

#### Original version (original\_fib\_list)

- Generates Fibonacci numbers
  - Converts each number to string inside the loop
  - Stores all values in a list
- More memory usage, mixed logic (generation + formatting)

#### Optimized version (optimized\_fib\_generator)

- Uses a **generator (yield)**
  - Produces numbers one-by-one
  - Conversion to string happens only when printing
- Less memory, cleaner design, reusable

## Demo

- Prints Fibonacci using both methods
- Output is the same

## Benchmark

- Measures **time** and **memory**
- Optimized version:
  - Uses less peak memory
  - Scales better for large n

## Key takeaway

- Generator = better performance + cleaner code
- Separate **logic (generation)** from **presentation (printing)**

## Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)

### PROMPT:

### Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)

#### ❖ Scenario

The Fibonacci logic is now required in multiple modules of an application.

#### ❖ Task Description

Use GitHub Copilot to generate a function-based Python program that:

- Uses a user-defined function to generate Fibonacci numbers
- Returns or prints the sequence up to n
- Includes meaningful comments (AI-assisted)

#### ❖ Expected Output

- Correct function-based Fibonacci implementation
- Screenshots documenting Copilot's function generation
- Sample test cases with outputs

### Code Explanation:

#### Optimization & Cleanup (Fibonacci)

### Original version (`original_fib_list`)

- Generates Fibonacci numbers
- Converts each number to string inside the loop
- Stores all values in a list
  - **More memory usage**, mixed logic (generation + formatting)

### Optimized version (`optimized_fib_generator`)

- Uses a **generator (yield)**
- Produces numbers one-by-one
- Conversion to string happens only when printing
  - **Less memory**, cleaner design, reusable

### Demo

- Prints Fibonacci using both methods
- Output is the same

### Benchmark

- Measures **time and memory**
- Optimized version:
  - Uses less peak memory
  - Scales better for large n

### Key takeaway

- Generator = better performance + cleaner code
- Separate **logic (generation)** from **presentation (printing)**

The screenshot shows a VS Code interface with several tabs at the top: ai.py, task2.py, task3.py (active), task4.py, task5.py, and cp.py. The task3.py tab contains the following Python code:

```
'''task3.py
Task 3: Modular Design Using AI Assistance (Fibonacci Using Functions)

This module provides a function-based implementation to generate Fibonacci numbers.
It supports returning the first N Fibonacci numbers (by count) or the sequence up to
a maximum value.

AI note: comments and docstrings were enhanced with GitHub Copilot assistance. "docstrings": Unknown word.

'''
from __future__ import annotations
from typing import List

def generate_fibonacci(n: int, *, up_to_value: bool = False) -> List[int]:
    """Generate a sequence of Fibonacci numbers.

    Parameters:
        n (int): The number of elements in the sequence.
        up_to_value (bool): If True, generates the sequence up to the value n.
    Returns:
        List[int]: The generated sequence of Fibonacci numbers.
    """
    if n == 0:
        return []
    elif n == 1:
        return [0]
    else:
        sequence = [0, 1]
        while len(sequence) < n:
            next_value = sequence[-1] + sequence[-2]
            if up_to_value and next_value >= n:
                break
            sequence.append(next_value)
        return sequence[:n] if not up_to_value else sequence
```

The terminal below shows command-line interactions:

```
PS C:\Users\preet\Downloads\pythonbasics> & C:/Users/preet/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/preet/Downloads/pythonbasics/task3.py
Example: first 7 Fibonacci numbers (count=7)
0, 1, 1, 2, 3, 5, 8

Example: Fibonacci numbers up to value 20 (up_to_value=True)
0, 1, 1, 2, 3, 5, 8, 13

Example: edge cases
n=0 -> []
n=1 -> [0]
```

## OUTPUT:

Example: first 7 Fibonacci numbers (count=7)

0, 1, 1, 2, 3, 5, 8

Example: Fibonacci numbers up to value 20 (up\_to\_value=True)

0, 1, 1, 2, 3, 5, 8, 13

Example: edge cases

n=0 -> []

n=1 -> [0]

All simple tests passed.

Notes: To document Copilot's function generation, open this file in VS Code, enable GitHub Copilot, trigger an inline suggestion for `generate\_fibonacci`, and take a screenshot of the suggestion popup or accepted code for your report.

## Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

## PROMPT:

### Task 4: Comparative Analysis – Procedural vs Modular Fibonacci Code

#### ❖ Scenario

You are participating in a code review session.

#### ❖ Task Description

Compare the Copilot-generated Fibonacci programs:

➤ Without functions (Task 1)

➤ With functions (Task 3)

➤ Analyze them in terms of:

- Code clarity

- Reusability

- Debugging ease

- Suitability for larger systems

#### ❖ Expected Output

Comparison table or short analytical report

```
ai.py task2.py task3.py task4.py x task5.py cp.py
task4.py > ...
1 '''Task 4: Comparative Analysis - Procedural vs Modular Fibonacci Code
2 ❖ Scenario
3 You are participating in a code review session.
4 ❖ Task Description
5 Compare the Copilot-generated Fibonacci programs:
6 ➤ Without functions (Task 1)
7 ➤ With functions (Task 3)
8 ➤ Analyze them in terms of:
9 ▪ Code clarity
10 ▪ Reusability
11 ▪ Debugging ease
12 ▪ Suitability for larger systems
13 ❖ Expected Output
14 Comparison table or short analytical report summarizing the strengths and weaknesses'''
15
16 OUTPUT = (
17     "Comparison - Procedural ('ai.py') vs Modular ('task3.py')\n"
18     "-----\n"
PRBLEMS DEBUG CONSOLE TERMINAL PORTS SPELL CHECKER
PS C:\Users\preet\Downloads\pythonbasics> & C:/Users/preet/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/preet/Downloads/pythonbasics/task3.py
PS C:\Users\preet\Downloads\pythonbasics> & C:/Users/preet/AppData/Local/Microsoft/WindowsApps/python3.12.exe c:/Users/preet/Downloads/pythonbasics/task4.py
Comparison - Procedural ('ai.py') vs Modular ('task3.py')
-----
Criterion: Code clarity
- ai.py: Linear script, minimal comments (implicit intent)
- task3.py: Clear functions, docstrings, type hints
```

OUTPUT:

Comparison — Procedural ('ai.py') vs Modular ('task3.py')

---

Criterion: Code clarity

- ai.py: Linear script, minimal comments (implicit intent)

- task3.py: Clear functions, docstrings, type hints

#### Criterion: Reusability

- ai.py: Tied to stdin/stdout and script flow — hard to reuse
- task3.py: Pure functions that return values, flexible options (`up\_to\_value`)

#### Criterion: Debugging ease

- ai.py: Must debug main flow; no isolated units or tests
- task3.py: Small, testable functions with explicit edge cases

#### Criterion: Suitability for larger systems

- ai.py: Poor — not composable or easily testable
- task3.py: Good — clear API, easy to extend and integrate

#### Short analytical summary:

- ai.py is acceptable for quick demos but lacks documentation and composability.
- task3.py follows best practices (functions, types, docstrings, tests) and is preferable for maintainability and production.
- Recommendation: prefer `task3.py`. Extract logic from `ai.py` if you need a reusable API.

## **Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)**

#### **PROMPT :**

Task 5: AI-Generated Iterative vs Recursive Fibonacci Approaches (Different Algorithmic Approaches for Fibonacci Series)

#### **❖ Scenario**

Your mentor wants to assess AI's understanding of different algorithmic

paradigms.

## ❖ Task Description

## Prompt GitHub Copilot to generate:

## An iterative Fibonacci implementation

## A recursive Fibonacci implementation

## ❖ Expected Output

- Two correct implementations
  - Explanation of execution flow for both
  - Comparison covering:
    - Time and space complexity
    - Performance for large n
    - When recursion should be avoided

## **OUTPUT:**

First 10 Fibonacci numbers (iterative):

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

First 10 Fibonacci numbers (recursive):

[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]

Timing for n=30 (naive recursive vs iterative):

naive recursive: value=832040, time=0.185535s

iterative: value=832040, time=0.000009s

memoized recursive for n=500 computed in 0.001015s (memoized)

## CODE EXPLANATION

### Task 5: Iterative vs Recursive Fibonacci

#### `fib_iterative(n)`

- Uses a loop to calculate Fibonacci
- Updates values using  $a, b = b, a + b$
- Time:  $O(n)$
- Space:  $O(1)$ 
  - ✓ Fast and memory-efficient

#### `fib_recursive(n)`

- Follows the formula  $F(n) = F(n-1) + F(n-2)$
- Repeats many calculations
- Time:  $O(2^n)$
- Space:  $O(n)$  (call stack)
  - ✗ Very slow for large n

#### `fib_recursive_memo(n)`

- Uses `@lru_cache` to store previous results
- Avoids repeated work
- Time:  $O(n)$
- Space:  $O(n)$ 
  - ✓ Fast but still uses recursion

## Demo & Timing

- Prints first 10 Fibonacci numbers
- Compares execution time of iterative vs recursive
- Shows memoized recursion is much faster than naive recursion

## Conclusion

- Iterative → best for performance

- **Naive recursive → only for learning**
- **Memoized recursive → good balance, but iterative is safest for large n**